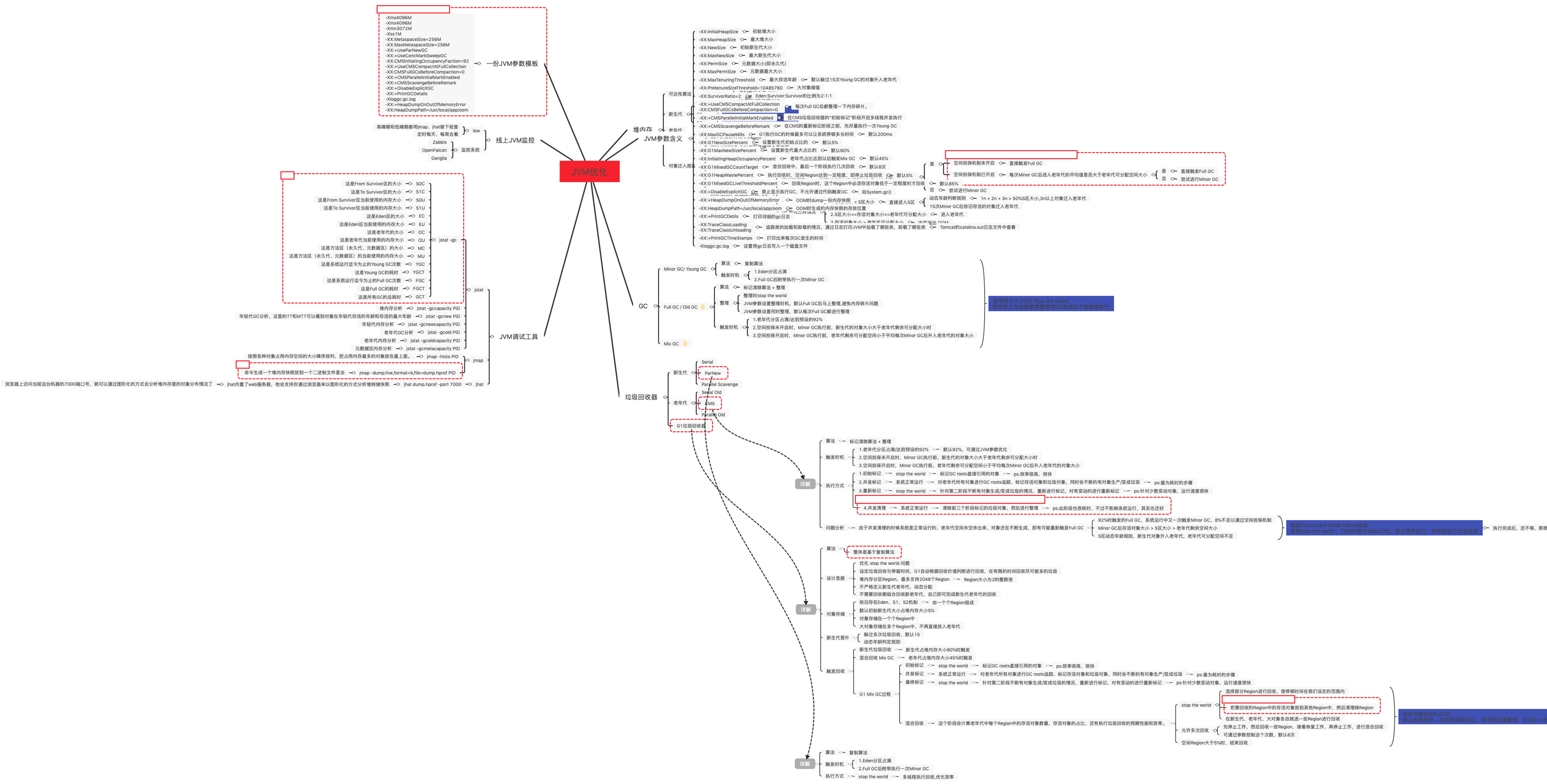
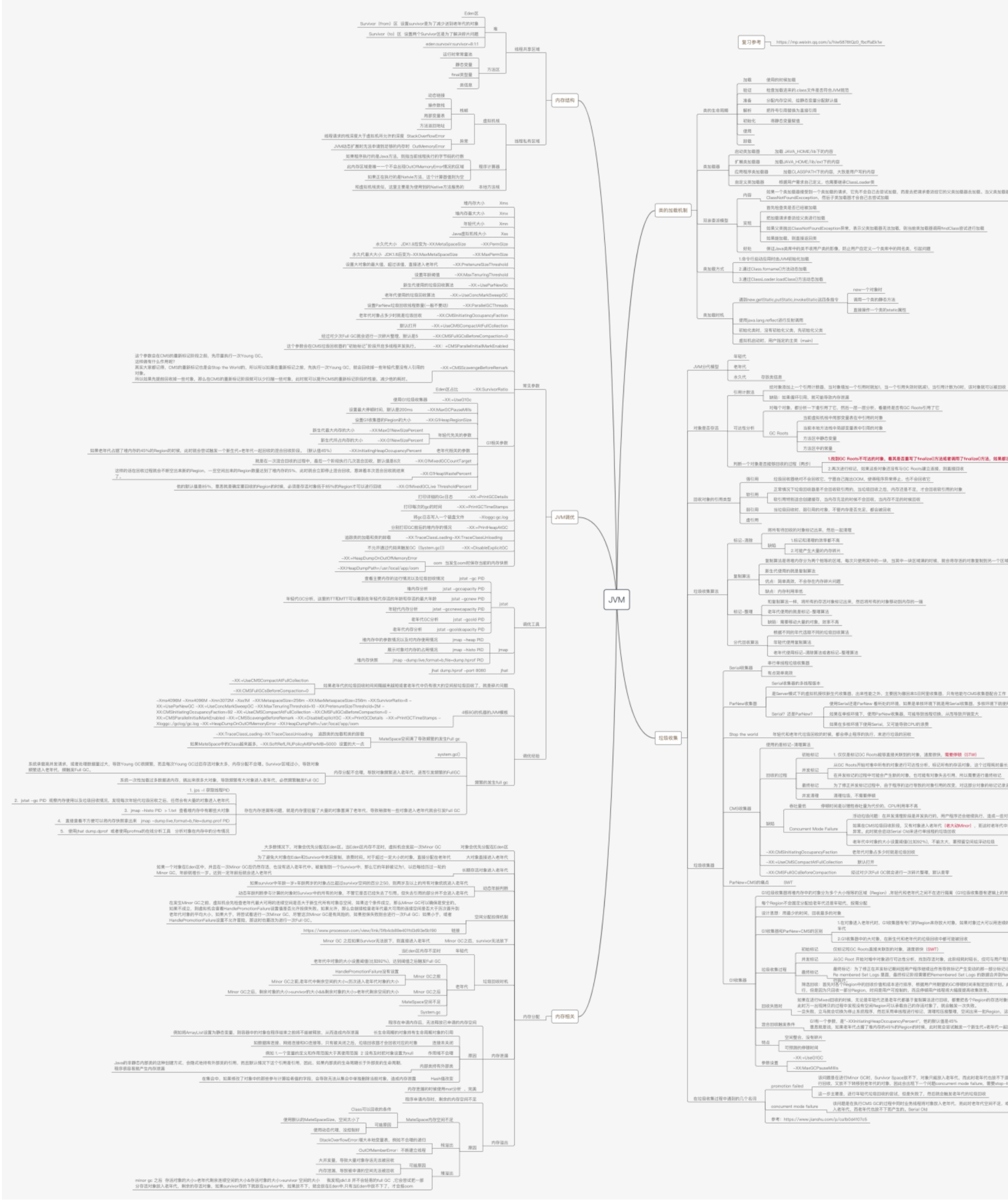
JVM


JVM优化
一份JVM参数模板
-Xms4096M
-Xmx4096M
-Xmn3072M
-Xss1M
-XX:MetaspaceSize=256M
-XX:MaxMetaspaceSize=256M
-XX:+UseParNewGC
-XX:+UseConcMarkSweepGC
-XX:CMSInitiatingOccupancyFaction=92
-XX:+UseCMSCompactAtFullCollection
-XX:CMSFullGCsBeforeCompaction=0
-XX:+CMSParallelInitialMarkEnabled
-XX:+CMSScavengeBeforeRemark
-XX:+DisableExplicitGC
-XX:+PrintGCDetails
-Xloggc:gc.log
-XX:+HeapDumpOnOutOfMemoryError
-XX:HeapDumpPath=/usr/local/app/oom
线上JVM监控
low
高峰期和低峰期都用jmap、jhat做下检查
定时每天、每周去看
监控系统
Zabbix
OpenFalcon
Ganglia
JVM调试工具
jstat
jstat -gc
S0C 这是From Survivor区的大小
S1C 这是To Survivor区的大小
S0U 这是From Survivor区当前使用的内存大小
S1U 这是To Survivor区当前使用的内存大小
EC 这是Eden区的大小
EU 这是Eden区当前使用的内存大小
OC 这是老年代的大小
OU 这是老年代当前使用的内存大小
MC 这是方法区（永久代、元数据区）的大小
MU 这是方法区（永久代、元数据区）的当前使用的内存大小
YGC 这是系统运行迄今为止的Young GC次数
YGCT 这是Young GC的耗时
FGC 这是系统运行迄今为止的Full GC次数
FGCT 这是Full GC的耗时
GCT 这是所有GC的总耗时
jstat -gccapacity PID 堆内存分析
jstat -gcnew PID 年轻代GC分析，这里的TT和MTT可以看到对象在年轻代存活的年龄和存活的最大年龄
jstat -gcnewcapacity PID 年轻代内存分析
jstat -gcold PID 老年代GC分析
jstat -gcoldcapacity PID 老年代内存分析
jstat -gcmetacapacity PID 元数据区内存分析
jmap
jmap -histo PID 按照各种对象占用内存空间的大小降序排列，把占用内存最多的对象放在最上面。
jmap -dump:live,format=b,file=dump.hprof PID 命令生成一个堆内存快照放到一个二进制文件里去
jhat
jhat dump.hprof -port 7000 jhat内置了web服务器，他会支持你通过浏览器来以图形化的方式分析堆转储快照 浏览器上访问当前这台机器的7000端口号，就可以通过图形化的方式去分析堆内存里的对象分布情况了
堆内存
可达性算法
新生代
老年代
对象迁入规则
JVM参数含义
-XX:InitialHeapSize 初始堆大小
-XX:MaxHeapSize 最大堆大小
-XX:NewSize 初始新生代大小
-XX:MaxNewSize 最大新生代大小
-XX:PermSize 元数据大小(即永久代)
-XX:MaxPermSize 元数据最大大小
-XX:MaxTenuringThreshold 最大存活年龄 默认躲过15次Young GC的对象升入老年代
-XX:PretenureSizeThreshold=10485760 大对象阈值
-XX:SurvivorRatio=2 Eden:Survivor:Survivor的比例为2:1:1
-XX:+UseCMSCompactAtFullCollection
-XX:CMSFullGCsBeforeCompaction=0 每次Full GC后都整理一下内存碎片。
-XX:+CMSParallelInitialMarkEnabled 在CMS垃圾回收器的"初始标记"阶段开启多线程并发执行
-XX:+CMSScavengeBeforeRemark 在CMS的重新标记阶段之前，先尽量执行一次Young GC
-XX:MaxGCPauseMills G1执行GC的时候最多可以让系统停顿多长时间 默认200ms
-XX:G1NewSizePercent 设置新生代初始占比的 默认5%
-XX:G1MaxNewSizePercent 设置新生代最大占比的 默认60%
-XX:InitiatingHeapOccupancyPercent 老年代占比达到以后触发Mix GC 默认45%
-XX:G1MixedGCCountTarget 混合回收中，最后一个阶段执行几次回收 默认8次
-XX:G1HeapWastePercent 执行回收时，空闲Region达到一定程度，即停止垃圾回收 默认5%
-XX:G1MixedGCLiveThresholdPercent 回收Region时，这个Region中必须存活对象低于一定程度时才回收 默认85%
-XX:+DisableExplicitGC 禁止显示执行GC，不允许通过代码触发GC 如System.gc()
-XX:+HeapDumpOnOutOfMemoryError OOM时dump一份内存快照
-XX:HeapDumpPath=/usr/local/app/oom OOM时生成的内存快照的存放位置
-XX:+PrintGCDetils 打印详细的gc日志
-XX:+TraceClassLoading
-XX:+TraceClassUnloading 追踪类的加载和卸载的情况，通过日志打印JVM中加载了哪些类，卸载了哪些类 Tomcat的catalina.out日志文件中查看
-XX:+PrintGCTimeStamps 打印出来每次GC发生的时间
-Xloggc:gc.log 设置将gc日志写入一个磁盘文件
是 空间担保机制未开启 直接触发Full GC
空间担保机制已开启 每次Minor GC后进入老年代的平均值是否大于老年代可分配空间大小
是 直接触发Full GC
否 尝试进行Minor GC
否 尝试进行Minor GC
< S区大小 直接进入S区
动态年龄判断规则 1n + 2n + 3n > 50%S区大小,3n以上对象迁入老年代
15次Minor GC后依旧存活的对象迁入老年代
2.S区大小<=存活对象大小<=老年代可分配大小 进入老年代
GC
Minor GC/ Young GC
算法 复制算法
触发时机
1.Eden分区占满
2.Full GC后附带执行一次Minor GC
Full GC / Old GC
算法 标记清除算法 + 整理
整理
整理时stop the world
JVM参数设置整理时机，默认Full GC后马上整理,避免内存碎片问题
JVM参数设置何时整理，默认每次Full GC都进行整理
触发时机
1.老年代分区占满/达到预设的92%
2.空间担保未开启时，Minor GC执行前，新生代的对象大小大于老年代剩余可分配大小时
3.空间担保开启时，Minor GC执行前，老年代剩余可分配空间小于平均每次Minor GC后升入老年代的对象大小
Mix GC
通常情况下,GC时 Stop the World
即所有工作线程都需要等待GC完成后才能继续执行
垃圾回收器
新生代
Serial
ParNew
Parallel Scavenge
老年代
Serial Old
CMS
Parallel Old
G1垃圾回收器
详解
算法 标记清除算法 + 整理
触发时机
1.老年代分区占满/达到预设的92% 默认92%，可通过JVM参数优化
2.空间担保未开启时，Minor GC执行前，新生代的对象大小大于老年代剩余可分配大小时
3.空间担保开启时，Minor GC执行前，老年代剩余可分配空间小于平均每次Minor GC后升入老年代的对象大小
执行方式
1.初始标记 stop the world 标记GC roots直接引用的对象 ps:效率很高，很快
2.并发标记 系统正常运行 对老年代所有对象进行GC roots追踪，标记存活对象和垃圾对象，同时会不断的有对象生产/变成垃圾 ps:最为耗时的步骤
3.重新标记 stop the world 针对第二阶段不断有对象生成/变成垃圾的情况，重新进行标记，对有变动的进行重新标记 ps:针对少数变动对象，运行速度很快
4.并发清理 系统正常运行 清除前三个阶段标记的垃圾对象，然后进行整理 ps:此阶段也很耗时，不过不影响系统运行，其实也还好
问题分析 由于并发清理的时候系统是正常运行的，老年代空间未空余出来，对象还在不断生成，即有可能重新触发Full GC
92%时触发的Full GC，系统运行中又一次触发Minor GC，8%不足以通过空间担保机制
Minor GC后存活对象大小 > S区大小 > 老年代剩余空间大小
S区动态年龄规则，新生代对象升入老年代，老年代可分配空间不足
触发Concurrent Mode Failure机制
直接stop the world ，CMS切换为Serial Old，禁止程序运行，单线程执行垃圾回收
执行完成后，还不够，那就
详解
算法 整体是基于复制算法
设计思路
优化 stop the world 问题
设定垃圾回收与停留时间，G1自动根据回收价值判断进行回收，在有限的时间回收尽可能多的垃圾
堆内存分区Region，最多支持2048个Region Region大小为2的整数倍
不严格定义新生代老年代，动态分配
不需要回收期组合回收新老年代，自己即可完成新生代老年代的回收
对象存储
依旧存在Eden、S1、S2机制 由一个个Region组成
默认初始新生代大小占堆内存大小5%
对象存储在一个个Region中
大对象存储在多个Region中，不再直接放入老年代
新生代晋升
躲过多次垃圾回收，默认15
动态年龄判定规则
触发回收
新生代垃圾回收 新生代占堆内存大小60%时触发
混合回收 Mix GC 老年代占堆内存大小45%时触发
G1 Mix GC过程
初始标记 stop the world 标记GC roots直接引用的对象 ps:效率很高，很快
并发标记 系统正常运行 对老年代所有对象进行GC roots追踪，标记存活对象和垃圾对象，同时会不断的有对象生产/变成垃圾 ps:最为耗时的步骤
最终标记 stop the world 针对第二阶段不断有对象生成/变成垃圾的情况，重新进行标记，对有变动的进行重新标记 ps:针对少数变动对象，运行速度很快
混合回收 这个阶段会计算老年代中每个Region中的存活对象数量，存活对象的占比，还有执行垃圾回收的预期性能和效率。
stop the world
选择部分Region进行回收，使停顿时间在我们设定的范围内
把要回收的Region中的存活对象放到其他Region中，然后清理掉Region
在新生代、老年代、大对象各自挑选一些Region进行回收
允许多次回收
先停止工作，然后回收一些Region，接着恢复工作，再停止工作，进行混合回收
可通过参数控制这个次数，默认8次
空闲Region大于5%时，结束回收
回收失败时的Full GC
停止系统程序，采用单线程标记、清理和压缩整理，空闲出一批
详解
算法 复制算法
触发时机
1.Eden分区占满
2.Full GC后附带执行一次Minor GC
执行方式 stop the world 多线程执行回收,优化效率



- 类加载过程

  加载 -> 验证 -> 准备 -> 解析 -> 初始化 -> 使用 -> 卸载

  （1）加载，将对应的“.class”字节码文件加载对应的类到内存里来

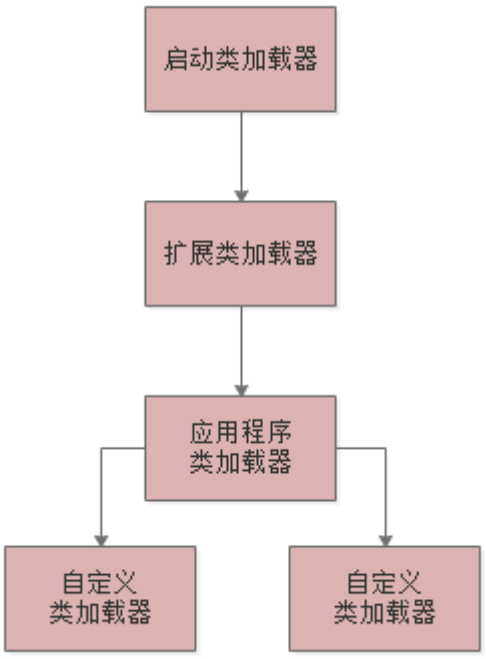
  （2）验证，把“.class”加载到内存里之后，必须先验证一下，校验他必须完全符合JVM规范，后续才能交给JVM来运行

  （3）准备，给类分配一定的内存空间，类变量（也就是static修饰的变量）分配内存空间，来一个默认的初始值

  （4）解析，把符号引用替换为直接引用

  （5）初始化，给变量赋值，static静态代码块也会在这个阶段执行，如果父类还没初始化，就先初始化它的父类

- 类加载器



（1）启动类加载器

Bootstrap ClassLoader，他主要是负责加载我们在机器上安装的Java目录下的核心类的，在你的Java安装目录下，就有一个“lib”目录，启动类加载器，去加载你的Java安装目录下的“lib”目录中的核心类库

（2）扩展类加载器

Extension ClassLoader，这个类加载器其实也是类似的，就是你的Java安装目录下，有一个“lib\ext”目录，这里面有一些类，就是需要使用这个类加载器来加载的，支撑你的系统的运行。

（3）应用程序类加载器

Application ClassLoader，这类加载器就负责去加载“ClassPath”环境变量所指定的路径中的类

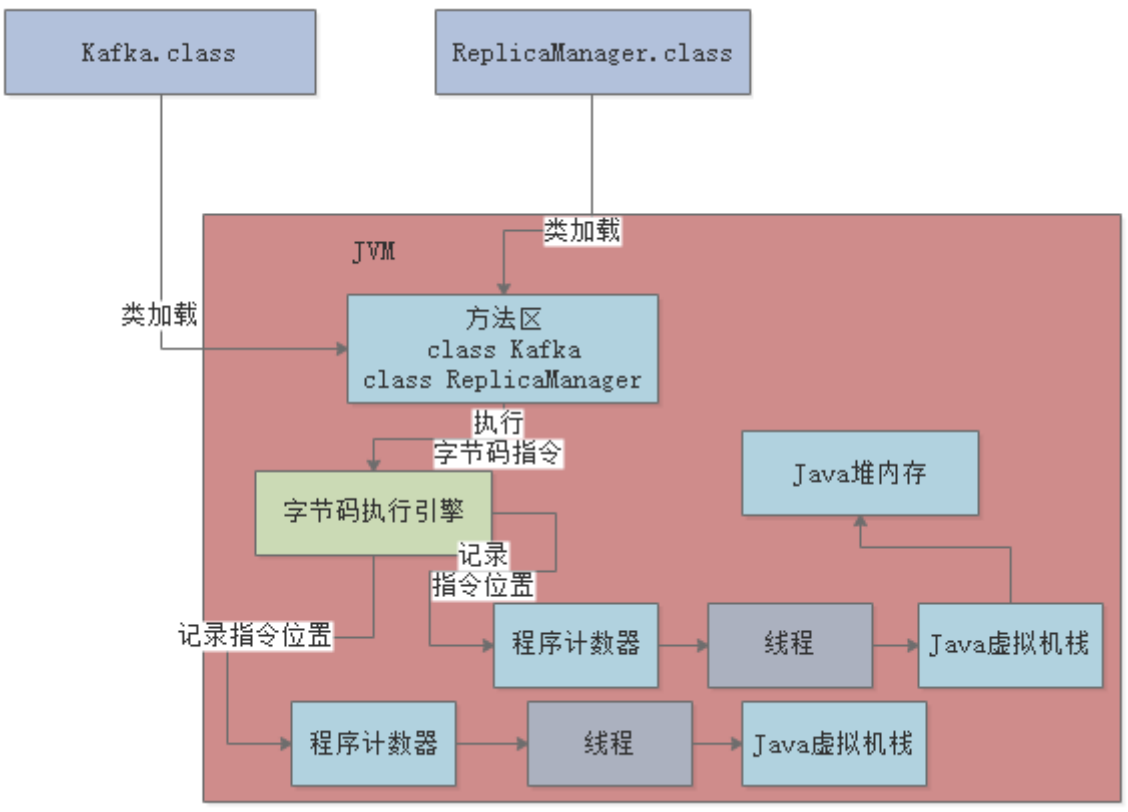
（4）自定义类加载器

除了上面那几种之外，还可以自定义类加载器，去根据你自己的需求加载你的类

- 双亲委派机制

比如你的JVM现在需要加载“ReplicaManager”类，此时应用程序类加载器会问问自己的爸爸，也就是扩展类加载器，你能加载到这个类吗？然后扩展类加载器直接问自己的爸爸，启动类加载器，你能加载到这个类吗？启动类加载器心想，我在Java安装目录下，没找到这个类啊，自己找去！然后，就下推加载权利给扩展类加载器这个儿子，结果扩展类加载器找了半天，也没找到自己负责的目录中有这个类。这时他很生气，说：明明就是你应用程序加载器自己负责的，你自己找去。然后应用程序类加载器在自己负责的范围内，比如就是你写好的那个系统打包成的jar包吧，一下子发现，就在这里！然后就自己把这个类加载到内存里去了。这就是所谓的双亲委派模型：先找父亲去加载，不行的话再由儿子来加载。这样的话，可以避免多层级的加载器结构重复加载某些类。

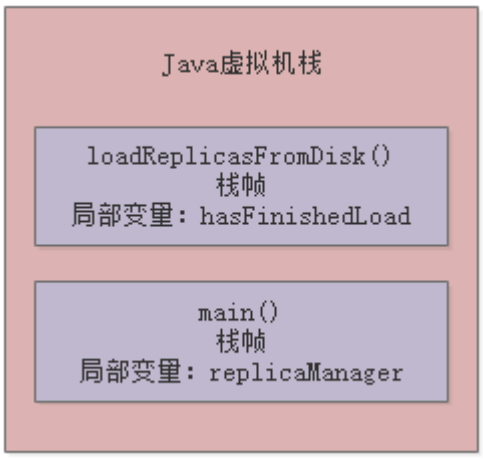
- JVM的内存区域



（1）方法区，Metaspace，元空间，主要是放从“.class”文件里加载进来的类，还会有一些类似常量池的东西放在这个区域里。

（2）程序计数器，来记录当前执行的字节码指令的位置的，也就是记录目前执行到了哪一条字节码指令

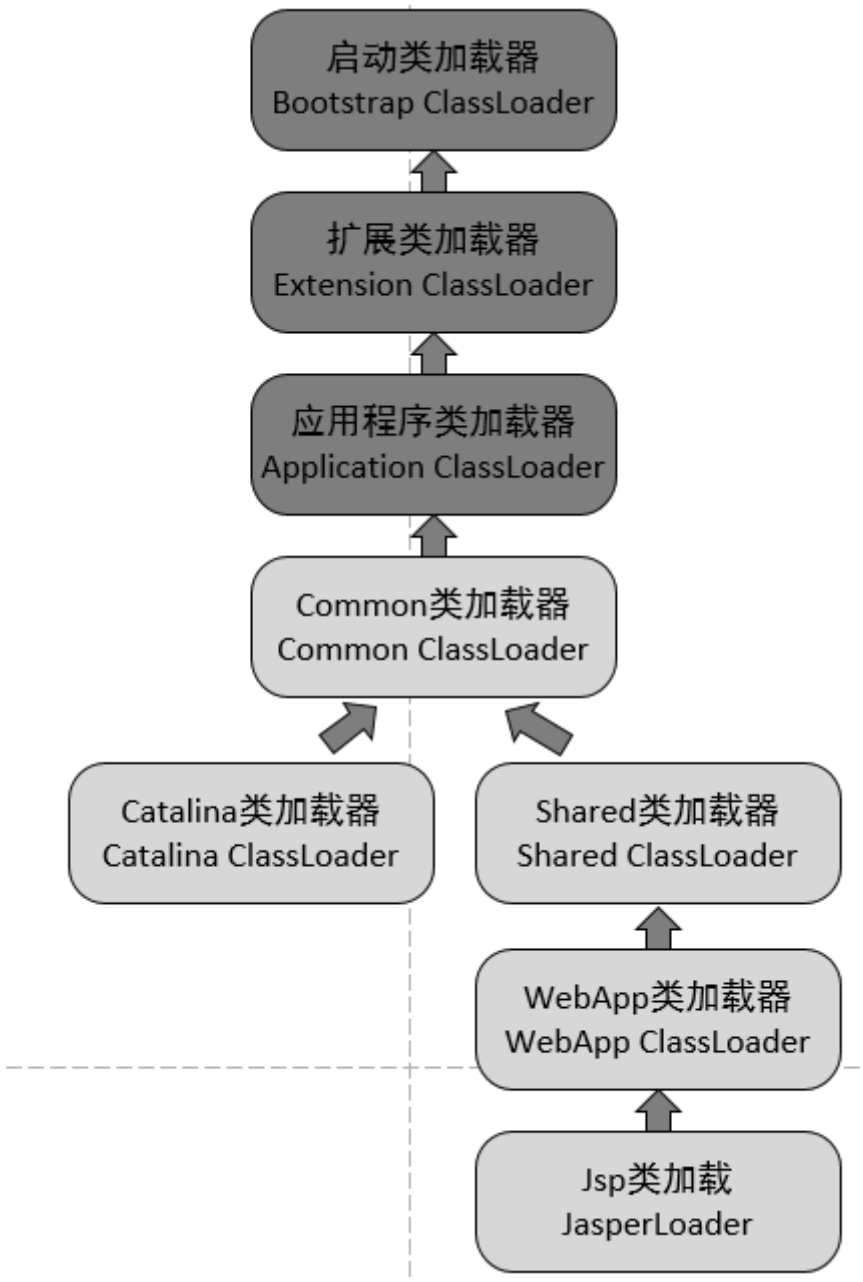
（3）Java虚拟机栈，JVM必须有一块区域是来保存每个方法内的局部变量等数据的，每个线程都有自己的Java虚拟机栈，比如这里的main线程就会有自己的一个Java虚拟机栈，用来存放自己执行的那些方法的局部变量。如果线程执行了一个方法，就会对这个方法调用创建对应的一个栈帧，栈帧里就有这个方法的局部变量表 、操作数栈、动态链接、方法出口等东西。

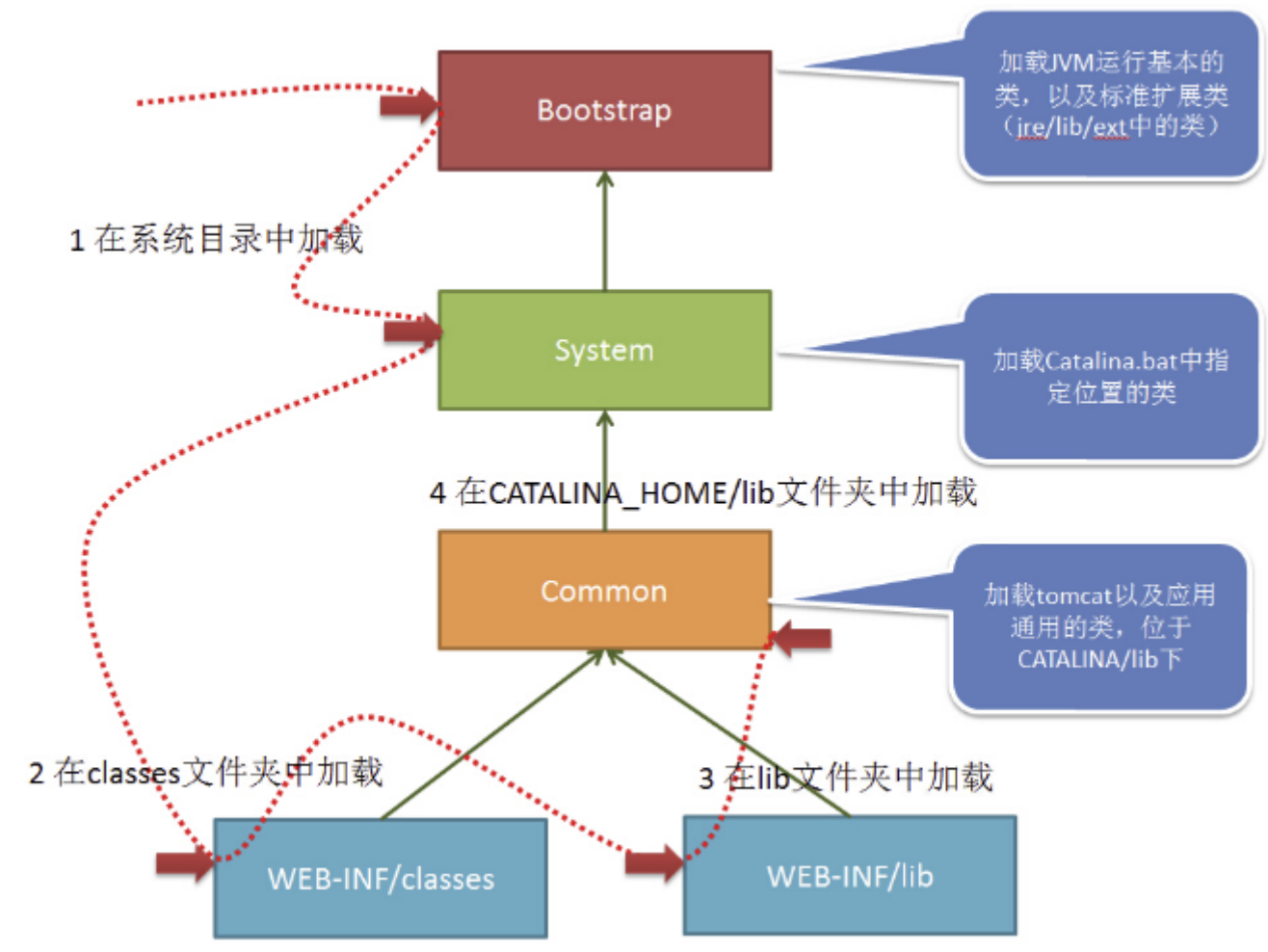


（4）Java堆内存，存放我们在代码中创建的各种对象的

- Tomcat类加载机制

Tomcat的类加载机制是违反了双亲委托原则的，对于一些未加载的非基础类(Object,String等)，各个web应用自己的类加载器(WebAppClassLoader)会优先加载，加载不到时再交给commonClassLoader走双亲委托。对于JVM来说：因此，按照这个过程可以想到，如果同样在CLASSPATH指定的目录中和自己工作目录中存放相同的class，会优先加载CLASSPATH目录中的文件。





（1）既然 Tomcat 不遵循双亲委派机制，那么如果我自己定义一个恶意的HashMap，会不会有风险呢?

显然不会有风险，如果有，Tomcat都运行这么多年了，那群Tomcat大神能不改进吗？ tomcat不遵循双亲委派机制，只是自定义的classLoader顺序不同，但顶层还是相同的，还是要去顶层请求classloader

（2）Tomcat是个web容器， 那么它要解决什么问题

1. 一个web容器可能需要部署两个应用程序，不同的应用程序可能会依赖同一个第三方类库的不同版本，不能要求同一个类库在同一个服务器只有一份，因此要保证每个应用程序的类库都是独立的，保证相互隔离。

2. 部署在同一个web容器中相同的类库相同的版本可以共享。否则，如果服务器有10个应用程序，那么要有10份相同的类库加载进虚拟机，这是扯淡的。

3. web容器也有自己依赖的类库，不能与应用程序的类库混淆。基于安全考虑，应该让容器的类库和程序的类库隔离开来。

4. web容器要支持jsp的修改，我们知道，jsp 文件最终也是要编译成class文件才能在虚拟机中运行，但程序运行后修改jsp已经是司空见惯的事情，否则要你何用？ 所以，web容器需要支持 jsp 修改后不用重启。

（3）Tomcat 如果使用默认的类加载机制行不行?

不行的。为什么？我们看，第一个问题，如果使用默认的类加载器机制，那么是无法加载两个相同类库的不同版本的，默认的类加器是不管你是什么版本的，只在乎你的全限定类名，并且只有一份。第二个问题，默认的类加载器是能够实现的，因为他的职责就是保证唯一性。第三个问题和第一个问题一样。我们再看第四个问题，我们想我们要怎么实现jsp文件的热修改，jsp 文件其实也就是class文件，那么如果修改了，但类名还是一样，类加载器会直接取方法区中已经存在的，修改后的jsp是不会重新加载的。那么怎么办呢？我们可以直接卸载掉这jsp文件的类加载器，所以你应该想到了，每个jsp文件对应一个唯一的类加载器，当一个jsp文件修改了，就直接卸载这个jsp类加载器。重新创建类加载器，重新加载jsp文件。

（4）Tomcat 如何实现自己独特的类加载机制

我们看到，前面3个类加载和默认的一致，CommonClassLoader、CatalinaClassLoader、SharedClassLoader和WebappClassLoader则是Tomcat自己定义的类加载器，它们分别加载/common/*、/server/*、/shared/*（在tomcat 6之后已经合并到根目录下的lib目录下）和/WebApp/WEB-INF/*中的Java类库。其中WebApp类加载器和Jsp类加载器通常会存在多个实例，每一个Web应用程序对应一个WebApp类加载器，每一个JSP文件对应一个Jsp类加载器。commonLoader：Tomcat最基本的类加载器，加载路径中的class可以被Tomcat容器本身以及各个Webapp访问，catalinaLoader：Tomcat容器私有的类加载器，加载路径中的class对于Webapp不可见，sharedLoader：各个Webapp共享的类加载器，加载路径中的class对于所有Webapp可见，但是对于Tomcat容器不可见，WebappClassLoader：各个Webapp私有的类加载器，加载路径中的class只对当前Webapp可见。从图中的委派关系中可以看出：CommonClassLoader能加载的类都可以被Catalina ClassLoader和SharedClassLoader使用，从而实现了公有类库的共用，而CatalinaClassLoader和Shared ClassLoader自己能加载的类则与对方相互隔离。WebAppClassLoader可以使用SharedClassLoader加载到的类，但各个WebAppClassLoader实例之间相互隔离。而JasperLoader的加载范围仅仅是这个JSP文件所编译出来的那一个.Class文件，它出现的目的就是为了被丢弃：当Web容器检测到JSP文件被修改时，会替换掉目前的JasperLoader的实例，并通过再建立一个新的Jsp类加载器来实现JSP文件的HotSwap功能。tomcat 为了实现隔离性，没有遵循双亲委派模型，每个webappClassLoader加载自己的目录下的class文件，不会传递给父类加载器。

（5）如果tomcat 的 Common ClassLoader 想加载 WebApp ClassLoader 中的类，该怎么办?

我们可以使用线程上下文类加载器实现，使用线程上下文加载器，可以让父类加载器请求子类加载器去完成类加载的动作。

- JVM分代模型：年轻代、老年代、永久代

（1）为什么要分成年轻代和老年代

因为这跟垃圾回收有关，对于年轻代里的对象，他们的特点是创建之后很快就会被回收，所以需要用一种垃圾回收算法。对于老年代里的对象，他们的特点是需要长期存在，所以需要另外一种垃圾回收算法，所以需要分成两个区域来放不同的对象。

（2）方法区内垃圾回收

首先该类的所有实例对象都已经从Java堆内存里被回收，其次加载这个类的ClassLoader已经被回收，最后，对该类的Class对象没有任何引用，满足上面三个条件就可以回收该类了。

（3）新生代垃圾回收条件

如果新生代我们预先分配的内存空间，几乎都被全部对象给占满了！此时假设我们代码继续运行，他需要在新生代里去分配一个对象，这个时候，就会触发一次新生代内存空间的垃圾回收，新生代内存空间的垃圾回收，也称之为“Minor GC”，有的时候我们也叫“Young GC”，他会尝试把新生代里那些没有人引用的垃圾对象，都给回收掉。

（4）跟JVM内存相关的几个核心参数图解

1. -Xms：Java堆内存的大小
2. -Xmx：Java堆内存的最大大小
3. -Xmn：Java堆内存中的新生代大小，扣除新生代剩下的就是老年代的内存大小了
4. -XX:PermSize：永久代大小
5. -XX:MaxPermSize：永久代最大大小
6. -Xss：每个线程的栈内存大小

-Xms和-Xmx，分别用于设置Java堆内存的刚开始的大小，以及允许扩张到的最大大小。对于这对参数，通常来说，都会设置为完全一样的大小。-Xmn，这个参数也是很常见的，他用来设置Java堆内存中的新生代的大小，然后扣除新生代大小之后的剩余内存就是给老年代的内存大小。-XX:PermSize和-XX:MaxPermSize，分别限定了永久代大小和永久代的最大大小。如果是JDK 1.8以后的版本，那么这俩参数被替换为了-XX:MetaspaceSize和-XX:MaxMetaspaceSize，但是大家至少得知道，这两个参数限定了永久代的大小。-Xss，这个参数限定了每个线程的栈内存大小。

（5）哪些对象能被回收

JVM中使用了一种可达性分析算法来判定哪些对象是可以被回收的，哪些对象是不可以被回收的。这个算法的意思，就是说对每个对象，都分析一下有谁在引用他，然后一层一层往上去判断，看是否有一个GC Roots。在JVM规范中， java虚拟机栈中的引用的对象，方法区中的类静态属性引用的对象（一般指被static修饰的对象，加载类的时候就加载到内存中），方法区中的常量引用的对象，本地方法栈中的JNI（native方法）引用的对象是可以作为GC Roots。finalize()方法可以拯救以下马上要被垃圾回收的对象。

（6）四种引用类型

强引用，只要是强引用的类型，那么垃圾回收的时候绝对不会去回收这个对象的

软引用，正常情况下垃圾回收是不会回收软引用对象的，但是如果你进行垃圾回收之后，发现内存空间还是不够存放新的对象，内存都快溢出了此时就会把这些软引用对象给回收掉，哪怕他被变量引用了，但是因为他是软引用，所以还是要回收。

弱引用，你这个弱引用就跟没引用是类似的，如果发生垃圾回收，就会把这个对象回收掉。

虚引用，这个大家其实暂时忽略他也行，因为很少用

（7）垃圾回收算法

复制算法，将存活的对象移动到一个地方，然后回收其他区域的垃圾对象。

标记整理算法，首先标记出来老年代当前存活的对象，这些对象可能是东一个西一个的，接着会让这些存活对象在内存里进行移动，把存活对象尽量都挪动到一边去，让存活对象紧凑的靠在一起，避免垃圾回收过后出现过多的内存碎片，然后再一次性把垃圾对象都回收掉。

（8）年轻代对象什么时候进入老年代

1、躲过15次GC之后进入老年代，可以通过JVM参数“-XX:MaxTenuringThreshold”来设置，默认是15岁

2、动态对象年龄判断，大致规则就是，假如说当前放对象的Survivor区域里，一批对象的总大小大于了这块Survivor区域的内存大小的50%，那么此时大于等于这批对象年龄的对象，就可以直接进入老年代了。假设这个图里的Survivor2区有两个对象，这俩对象的年龄一样，都是2岁，然后俩对象加起来对象超过了50MB，超过了Survivor2区的100MB内存大小的一半了，这个时候，Survivor2区里的大于等于2岁的对象，就要全部进入老年代里去。实际这个规则运行的时候是如下的逻辑：年龄1+年龄2+年龄n的多个年龄对象总和超过了Survivor区域的50%，此时就会把年龄n以上的对象都放入老年代。

3、大对象直接进入老年代，有一个JVM参数，就是“-XX:PretenureSizeThreshold”，可以把他的值设置为字节数，比如“1048576”字节，就是1MB。如果你要创建一个大于这个大小的对象，比如一个超大的数组，或者是别的啥东西，此时就直接把这个大对象放到老年代里去。压根儿不会经过新生代。之所以这么做，就是要避免新生代里出现那种大对象，然后屡次躲过GC，还得把他在两个Survivor区域里来回复制多次之后才能进入老年代。

4、Minor GC后的对象太多无法放入Survivor区，这个时候就必须得把这些对象直接转移到老年代去

（9）老年代空间分配担保规则

如果新生代里有大量对象存活下来，确实是自己的Survivor区放不下了，必须转移到老年代去，那么如果老年代里空间也不够放这些对象呢？首先，在执行任何一次Minor GC之前，JVM会先检查一下老年代可用的可用内存空间，是否大于新生代所有对象的总大小。为啥检查这个呢？因为最极端的情况下，可能新生代Minor GC过后，所有对象都存活下来了，那岂不是新生代所有对象全部要进入老年代？如果说发现老年代的内存大小是大于新生代所有对象的，此时就可以放心大胆的对新生代发起一次Minor GC了，因为即使Minor GC之后所有对象都存活，Survivor区放不下了，也可以转移到老年代去。但是假如执行Minor GC之前，发现老年代的可用内存已经小于了新生代的全部对象大小了，那么这个时候是不是有可能在Minor GC之后新生代的对象全部存活下来，然后全部需要转移到老年代去，但是老年代空间又不够？理论上，是有这种可能的，所以假如Minor GC之前，发现老年代的可用内存已经小于了新生代的全部对象大小了，就会看一个“-XX:-HandlePromotionFailure”的参数是否设置了，如果有这个参数，那么就会继续尝试进行下一步判断，下一步判断，就是看看老年代的内存大小，是否大于之前每一次Minor GC后进入老年代的对象的平均大小。如果上面那个步骤判断失败了，或者是“-XX:-HandlePromotionFailure”参数没设置，此时就会直接触发一次“FullGC”，就是对老年代进行垃圾回收，尽量腾出来一些内存空间，然后再执行Minor GC。如果上面两个步骤都判断成功了，那么就是说可以冒点风险尝试一下Minor GC。此时进

行Minor GC有几种可能。第一种可能，Minor GC过后，剩余的存活对象的大小，是小于Survivor区的大小的，那么此时存活对象进入Survivor区域即可。第二种可能，Minor GC过后，剩余的存活对象的大小，是大于 Survivor区域的大小，但是是小于老年代可用内存大小的，此时就直接进入老年代即可。第三种可能，很不幸，Minor GC过后，剩余的存活对象的大小，大于了Survivor区域的大小，也大于了老年代可用内存的大小。此时老年代都放不下这些存活对象了，就会发生“Handle Promotion Failure”的情况，这个时候就会触发一次“Full GC”。Full GC就是对老年代进行垃圾回收，同时也一般会对新生代进行垃圾回收。如果要是Full GC过后，老年代还是没有足够的空间存放Minor GC过后的剩余存活对象，那么此时就会导致所谓的“OOM”内存溢出了。

- 垃圾收集器

（1）Serial和Serial Old垃圾回收器：分别用来回收新生代和老年代的垃圾对象，工作原理就是单线程运行，垃圾回收的时候会停止我们自己写的系统的其他工作线程，让我们系统直接卡死不动，然后让他们垃圾回收，这个现在一般写后台Java系统几乎不用。

（2）ParNew和CMS垃圾回收器：ParNew现在一般都是用在新生代的垃圾回收器，CMS是用在老年代的垃圾回收器，他们都是多线程并发的机制，性能更好，现在一般是线上生产系统的标配组合。新生代的ParNew垃圾回收器主打的就是多线程垃圾回收机制，另外一种Serial垃圾回收器主打的是单线程垃圾回收，他们俩都是回收新生代的，唯一的区别就是单线程和多线程的区别，但是垃圾回收算法是完全一样的。ParNew垃圾回收器如果一旦在合适的时机执行Minor GC的时候，就会把系统程序的工作线程全部停掉，禁止程序继续运行创建新的对象，然后自己就用多个垃圾回收线程去进行垃圾回收，回收的机制和算法就跟之前说的是一样的。ParNew垃圾回收器，默认给自己设置的垃圾回收线程的数量就是跟CPU的核数是一样的。但是如果你一定要自己调节ParNew的垃圾回收线程数量，也是可以的，使用“-XX:ParallelGCThreads”参数即可，通过他可以设置线程的数量。

（3）G1垃圾回收器：统一收集新生代 和老年代，采用了更加优秀的算法和设计机制

- CMS垃圾收集器

一般老年代我们选择的垃圾回收器是CMS，他采用的是标记清理算法，其实非常简单，就是先用之前文章里讲过的标记方法去标记出哪些对象是垃圾对象，然后就把这些垃圾对象清理掉。所谓的标记-清理算法，其实就是我们之前给大家讲过的一个算法，先通过追踪GC Roots的方法，看看各个对象是否被GC Roots给引用了，如果是的话，那就是存活对象，否则就是垃圾对象。先将垃圾对象都标记出来，然后一次性把垃圾对象都回收掉。这种方法其实最大的问题，就是会造成很多内存碎片。

CMS垃圾回收器采取的是垃圾回收线程和系统工作线程尽量同时执行的模式来处理的。

CMS在执行一次垃圾回收的过程一共分为4个阶段：

1. 初始标记，标记出来所有GC Roots直接引用的对象，虽然说要造成“Stop the World”暂停一切工作线程，但是其实影响不大，因为他的速度很快，仅仅标记GC Roots直接引用的那些对象罢了。

2. 并发标记，这个阶段会让系统线程可以随意创建各种新对象，继续运行，在运行期间可能会创建新的存活对象，也可能会让部分存活对象失去引用，变成垃圾对象。在这个过程中，垃圾回收线程，会尽可能的对已有的对象进行GC Roots追踪。但是这个过程中，在进行并发标记的时候，系统程序会不停的工作，他可能会各种创建出来新的对象，部分对象可能成为垃圾。对老年代所有对象进行GC Roots追踪，其实是最耗时的，他需要追踪所有对象是否从根源上被GC Roots引用了，但是这个最耗时的阶段，是跟系统程序并发运行的，所以其实这个阶段不会对系统运行造成影响的。

3. 重新标记，因为第二阶段里，你一边标记存活对象和垃圾对象，一边系统在不停运行创建新对象，让老对象变成垃圾，所以第二阶段结束之后，绝对会有很多存活对象和垃圾对象，是之前第二阶段没标记出来的，所以此时进入第三阶段，要继续让系统程序停下来，再次进入“Stop the World”阶段。然后重新标记下在第二阶段里新创建的一些对象，还有一些已有对象可能失去引用变成垃圾的情况。这个重新标记的阶段，是速度很快的，他其实就是对在第二阶段中被系统程序运行变动过的少数对象进行标记，所以运行速度很快。

4. 并发清理，这个阶段就是让系统程序随意运行，然后他来清理掉之前标记为垃圾的对象即可。这个阶段其实是很耗时的，因为需要进行对象的清理，但是他也是跟系统程序并发运行的，所以其实也不影响系统程序的执行。

Concurrent Mode Failure问题：为了保证在CMS垃圾回收期间，还有一定的内存空间让一些对象可以进入老年代，一般会预留一些空间。CMS垃圾回收的触发时机，其中有一个就是当老年代内存占用达到一定比例了，就自动执行GC。“-XX:CMSInitiatingOccupancyFaction”参数可以用来设置老年代占用多少比例的时候触发CMS垃圾回收，JDK 1.6里面默认的值是92%。也就是说，老年代占用了92%空间了，就自动进行CMS垃圾回收，预留8%的空间给并发回收期间，系统程序把一些新对象放入老年代中。那么如果CMS垃圾回收期间，系统程序要放入老年代的对象大于了可用内存空间，此时会如何？这个时候，会发生Concurrent Mode Failure，就是说并发垃圾回收失败了，我一边回收，你一边把对象放入老年代，内存都不够了。此时就会自动用“Serial Old”垃圾回收器替代CMS，就是直接强行把系统程序“Stop the World”，重新进行长时间的GC Roots追踪，标记出来全部垃圾对象，不允许新的对象产生。然后一次性把垃圾对象都回收掉，完事儿了再恢复系统线程。

内存碎片清理：CMS有一个参数是“-XX:+UseCMSCompactAtFullCollection”，在Full GC之后要再次进行“Stop the World”，停止工作线程，然后进行碎片整理，就是把存活对象挪到一起，空出来大片连续内存空间，避免内存碎片。还有一个参数是“-XX:CMSFullGCsBeforeCompaction”，这个意思是执行多少次Full GC之后再执行一次内存碎片整理的工作，默认是0，意思就是每次Full GC之后都会进行一次内存整理。

为啥老年代的Full GC要比新生代的Minor GC慢很多倍，一般在10倍以上？新生代执行速度其实很快，因为直接从GC Roots出发就追踪哪些对象是活的就行了，新生代存活对象是很少的，这个速度是极快的，不需要追踪多少对象。然后直接把存活对象放入Survivor中，就一次性直接回收Eden和之前使用的Survivor了。但是CMS的Full GC呢？在并发标记阶段，他需要去追踪所有存活对象，老年代存活对象很多，这个过程就会很慢；其次并发清理阶段，他不是一次性回收一大片内存，而是找到零零散散在各个地方的垃圾对象，速度也很慢；最后完事儿了，还得执行一次内存碎片整理，把大量的存活对象给挪在一起，空出来连续内存空间，这个过程还得“Stop theWorld”，那就更慢了。万一并发清理期间，剩余内存空间不足以存放要进入老年代的对象了，引发了“Concurrent Mode Failure”问题，那更是麻烦，还得立马用“Serial Old”垃圾回收器，“Stop the World”之后慢慢重新来一遍回收的过程，这更是耗时了。

“-XX:+CMSParallelInitialMarkEnabled”，这个参数会在CMS垃圾回收器的“初始标记”阶段开启多线程并发执行，初始标记阶段，是会进行Stop the World的，会导致系统停顿，所以这个阶段开启多线程并发之后，可以尽可能优化这个阶段的性能，减少Stop the World的时间。

“-XX:+CMSScavengeBeforeRemark”，这个参数会在CMS的重新标记阶段之前，先尽量执行一次Young GC。CMS的重新标记也是会Stop the World的，所以所以如果在重新标记之前，先执行一次Young GC，就会回收掉一些年轻代里没有人引用的对象。所以如果先提前回收掉一些对象，那么在CMS的重新标记阶段就可以少扫描一些对象，此时就可以提升CMS的重新标记阶段的性能，减少他的耗时。

- 老年代GC的时机

（1）老年代可用内存小于新生代全部对象的大小，如果没开启空间担保参数，会直接触发Full GC，所以一般空间担保参数都会打开；

（2）老年代可用内存小于历次新生代GC后进入老年代的平均对象大小，此时会提前Full GC

（3）新生代Minor GC后的存活对象大于Survivor，那么就会进入老年代，此时老年代内存不足

（4）“-XX:CMSInitiatingOccupancyFaction”参数，如果老年代可用内存大于历次新生代GC后进入老年代的对象平均大小，但是老年代已经使用的内存空间超过了这个参数指定的比例，也会自动触发Full GC

- G1垃圾收集器

把Java堆内存拆分为多个大小相等的Region，G1最大的一个特点，就是可以让我们设置一个垃圾回收的预期停顿时间，也就是说比如我们可以指定：希望G1在垃圾回收的时候，可以保证，在1小时内由G1垃圾回收导致的“Stop the World”时间，也就是系统停顿的时间，不能超过1分钟。其实G1如果要做到这一点，他就必须要追踪每个Region里的回收价值，啥叫做回收价值呢？他必须搞清楚每个Region里的对象有多少是垃圾，如果对这个Region进行垃圾回收，需要耗费多长时间，可以回收掉多少垃圾？G1可以做到让你来设定垃圾回收对系统的影响，他自己通过把内存拆分为大量小Region，以及追踪每个Region中可以回收的对象大小和预估时间，最后在垃圾回收的时候，尽量把垃圾回收对系统造成的影响控制在你指定的时间范围内，同时在有限的时间内尽量回收尽可能多的垃圾对象。这就是G1的核心设计思路。

在G1中，每一个Region时可能属于新生代，但是也可能属于老年代的，刚开始Region可能谁都不属于，然后接着就分配给了新生代，然后放了很多属于新生代的对象，接着就触发了垃圾回收这个Region，然后下一次同一个Region可能又被分配了老年代了，用来放老年代的长生存周期的对象，所以其实在G1对应的内存模型中，Region随时会属于新生代也会属于老年代，所以没有所谓新生代给多少内存，老年代给多少内存这一说了。

到底有多少个Region呢？每个Region的大小是多大呢？其实这个默认情况下自动计算和设置的，我们可以给整个堆内存设置一个大小，比如说用“-Xms”和“-Xmx”来设置堆内存的大小。然后JVM启动的时候一旦发现你使用的是G1垃圾回收器，可以使用“-XX:+UseG1GC”来指定使用G1垃圾回收器，此时会自动用堆大小除以2048。因为JVM最多可以有2048个Region，然后Region的大小必须是2的倍数，比如说1MB、2MB、4MB之类的。如果通过手动方式来指定，则是“-XX:G1HeapRegionSize”。刚开始的时候，默认新生代对堆内存的占比是5%，也就是占据200MB左右的内存，对应大概是100个Region，这个是可以通过“-XX:G1NewSizePercent”来设置新生代初始占比的，其实维持这个默认值即可。因为在系统运行中，JVM其实会不停的给新生代增加更多的Region，但是最多新生代的占比不会超过60%，可以通过“-XX:G1MaxNewSizePercent”。而且一旦Region进行了垃圾回收，此时新生代的Region数量还会减少，这些其实都是动态的。其实在G1中虽然把内存划分为了很多的 Region，但是其实还是有新生代、老年代的区分，而且新生代里还是有Eden和Survivor的划分的，大家应该还记得之前说过的一个新生代的参数，“-XX:SurvivorRatio=8”，所以这里还是可以区分出来属于新生代的Region里哪些属于Eden，哪些属于Survivor。既然G1的新生代也有Eden和Survivor的区分，那么触发垃圾回收的机制都是类似的，随着不停的在新生代的Eden对应的Region中放对象，JVM就会不停的给新生代加入更多的Region，直到新生代占据堆大小的最大比例60%。一旦新生代达到了设定的占据堆内存的最大大小60%，比如都有1200个Region了，里面的Eden可能占据了1000个Region，每个Survivor是100个Region，而且Eden区还占满了对象，这个时候还是会触发新生代的GC，G1就会用之前说过的复制算法来进行垃圾回收，进入一个“Stop the World”状态，然后把Eden对应的Region中的存活对象放入S1对应的Region中，接着回收掉Eden对应的Region中的垃圾对象。但是这个过程跟之前是有区别的，因为G1是可以设定目标GC停顿时间的，也就是G1执行GC的时候最多可以让系统停顿多长时间，可以通过“-XX:MaxGCPauseMills”参数来设定，默认值是200ms。那么G1就会通过之前说的，对每个Region追踪回收他需要多少时间，可以回收多少对象来选择回收一部分的Region，保证GC停顿时间控制在指定范围内，尽可能多的回收掉一些对象。

大对象Region：G1提供了专门的Region来存放大对象，而不是让大对象进入老年代的Region中。在G1中，大对象的判定规则就是一个大对象超过了一个Region大小的50%，比如按照上面算的，每个Region是2MB，只要一个大对象超过了1MB，就会被放入大对象专门的Region中。而且一个大对象如果太大，可能会横跨多个Region来存放。那堆内存里哪些Region用来存放大对象啊？很简单，之前说过了，在G1里，新生代和老年代的Region是不停的变化的，比如新生代现在占据了1200个Region，但是一次垃圾回收之后，就让里面1000个Region都空了，此时那1000个Region就可以不属于新生代了，里面很多Region可以用来存放大对象。那么还有人会问了，大对象既然不属于新生代和老年代，那么什么时候会触发垃圾回收呢？也很简单，其实新生代、老年代在回收的时候，会顺带带着大对象Region一起回收，所以这就是在G1内存模型下对大对象的分配和回收的策略。

G1的动态内存管理策略，他会根据情况动态的把Region分配给新生代、Eden、Survivor、老年代和大对象，但是新生代和老年代有一个各自的最大占比，然后在新生代的Eden满的时候，触发新生代垃圾回收。新生代的垃圾回收还是采用了复制算法，只不过会考虑预设GC停顿时间，保证垃圾回收的停顿时间不能超过预设时间，因此会挑选一些Region来进行垃圾回收。然后跟之前说的一样，如果一些对象在新生代熬过了一定次数的GC，或者是触发了动态年龄判定规则，或者是存活对象在Survivor放不下了，都会让对象进入老年代中。而大对象则是进入单独的大对象Region，不再进入老年代。所以实际上在G1中，还是会存在新生代的对象慢慢会因为各种情况进入老年代的。

什么时候触发新生代+老年代的混合垃圾回收？G1有一个参数，是“-XX:InitiatingHeapOccupancyPercent”，他的默认值是45%，意思就是说，如果老年代占据了堆内存的45%的Region的时候，此时就会尝试触发一个新生代+老年代一起回收的混合回收阶段。比如按照我们之前说的，堆内存有2048个Region，如果老年代占据了其中45%的Region，也就是接近1000个Region的时候，就会开始触发一个混合回收。

G1垃圾回收过程：

（1）初始标记，这个过程是需要进入“Stop the World”的，仅仅只是标记一下GC Roots直接能引用的对象，这个过程速度是很快的。

（2）并发标记，这个阶段会允许系统程序的运行，同时进行GC Roots追踪，从GC Roots开始追踪所有的存活对象，这个并发标记阶段还是很耗时的，因为要追踪全部的存活对象。但是这个阶段是可以跟系统程序并发运行的，所以对系统程序的影响不太大。而且JVM会对并发标记阶段对对象做出的一些修改记录起来，比如说哪个对象被新建了，哪个对象失去了引用。

（3）最终标记阶段，这个阶段会进入“Stop the World”，系统程序是禁止运行的，但是会根据并发标记 阶段记录的那些对象修改，最终标记一下有哪些存活对象，有哪些是垃圾对象。

（4）“混合回收“阶段，这个阶段会计算老年代中每个Region中的存活对象数量，存活对象的占比，还有执行垃圾回收的预期性能和效率。接着会停止系统程序，然后全力以赴尽快进行垃圾回收，此时会选择部分Region进行回收，因为必须让垃圾回收的停顿时间控制在我们指定的范围内。比如说老年代此时有1000个Region都满了，但是因为根据预定目标，本次垃圾回收可能只能停顿200毫秒，那么通过之前的计算得知，可能回收其中800个Region刚好需要200ms，那么就只会回收800个Region，把GC导致的停顿时间控制在我们指定的范围内。

到底是回收这些区域的哪些Region呢？因为我们设定了对GC停顿时间的目标，所以说他会从新生代、老年代、大对象里各自挑选一些Region，保证用指定的时间（比如200ms）回收尽可能多的垃圾，这就是所谓的混合回收。

G1垃圾回收器的一些参数：

-XX:G1MixedGCCountTarget”参数，就是在一次混合回收的过程中，最后一个阶段执行几次混合回收，默认值是8次，意味着最后一个阶段，先停止系统运行，混合回收一些Region，再恢复系统运行，接着再次禁止系统运行，混合回收一些Region，反复8次。假设一次混合回收预期要回收掉一共有160个Region，那么此时第一次混合回收，会回收掉一些Region，比如就是 20个Region。接着恢复系统运行一会儿，然后再执行一次“混合回收”，如下图，再次回收掉20个Region。如此反复执行8次混合回收阶段之后 ，不就把预订的160个Region都回收掉了？而且还把系统停顿时间控制在指定范围内。那么为什么要反复回收多次呢？因为你停止系统一会儿，回收掉一些Region，再让系统运行一会儿，然后再次停止系统一会儿，再次回收掉一些Region，这样可以尽可能让系统不要停顿时间过长，可以在多次回收的间隙，也运行一下。

“-XX:G1HeapWastePercent”，默认值是5%，在混合回收的时候，对Region回收都是基于复制算法进行的，都是把要回收的Region里的存活对象放入其他Region，然后这个Region中的垃圾对象全部清理掉，这样的话在回收过程就会不断空出来新的Region，一旦空闲出来的Region数量达到了堆内存的5%，此时就会 立即停止混合回收，意味着本次混合回收就结束了。而且从这里也能看出来G1整体是基于复制算法进行Region垃圾回收的，不会出现内存碎片的问题，不需要像CMS那样标记-清理之后，再进行内存碎片的整理。

“-XX:G1MixedGCLiveThresholdPercent”，他的默认值是85%，意思就是确定要回收的Region的时候，必须是存活对象低于85%的Region才可以进行回收，否则要是一个Region的存活对象多余85%，你还回收他干什么？这个时候要把85%的对象都拷贝到别的Region，这个成本是很高的。

回收失败时的Full GC：如果在进行Mixed回收的时候，无论是年轻代还是老年代都基于复制算法进行回收，都要把各个Region的存活对象拷贝到别的Region里去，此时万一出现拷贝的过程中发现没有空闲Region可以承载自己的存活对象了，就会触发 一次失败。一旦失败，立马就会切换为停止系统程序，然后采用单线程进行标记、清理和压缩整理，空闲出来一批Region，这个过程是极慢极慢的。

新生代什么时候回收？他会根据你预设的gc停顿时间，给新生代分配一些Region，然后到一定程度就触发gc，并且把gc时间控制在预设范围内，尽量避免一次性回收过多的Region导致gc停顿时间超出预期。“-XX:MaxGCPauseMills”参数，如果这个参数设置的小了，那么说明每次gc停顿时间可能特别短，此时G1一旦发现你对几十个Region占满了就立即触发新生代gc，然后gc频率特别频繁，虽然每次gc时间很短。如果这个参数设置大了呢？那么可能G1会允许你不停的在新生代理分配新的对象，然后积累了很多对象了，再一次性回收几百个Region，此时可能一次GC停顿时间就会达到几百毫秒，但是GC的频率很低。

- jstat（定位young gc和Full gc）

分析运行中的系统，他的对象增长的速率，Young GC的触发频率，Young GC的耗时，每次Young GC后有多少对象是存活下来的，每次Young GC过后有多少对象进入了老年代，老年代对象增长的速率，Full GC的触发频率，Full GC的耗时。

jstat -gc PID

1. S0C：这是From Survivor区的大小
2. S1C：这是To Survivor区的大小
3. S0U：这是From Survivor区当前使用的内存大小
4. S1U：这是To Survivor区当前使用的内存大小
5. EC：这是Eden区的大小
6. EU：这是Eden区当前使用的内存大小
7. OC：这是老年代的大小
8. OU：这是老年代当前使用的内存大小
9. MC：这是方法区（永久代、元数据区）的大小
10. MU：这是方法区（永久代、元数据区）的当前使用的内存大小
11. YGC：这是系统运行迄今为止的Young GC次数
12. YGCT：这是Young GC的耗时
13. FGC：这是系统运行迄今为止的Full GC次数
14. FGCT：这是Full GC的耗时
15. GCT：这是所有GC的总耗时

其他的jstat命令

1. jstat -gccapacity PID：堆内存分析
2. jstat -gcnew PID：年轻代GC分析，这里的TT和MTT可以看到对象在年轻代存活的年龄和存活的最大年龄
3. jstat -gcnewcapacity PID：年轻代内存分析
4. jstat -gcold PID：老年代GC分析
5. jstat -gcoldcapacity PID：老年代内存分析
6. jstat -gcmetacapacity PID：元数据区内存分析

新生代对象增长的速率：jstat -gc PID 1000 10

- jmap

  jmap生成堆内存转储快照，jmap -dump:live,format=b,file=dump.hprof PID

  jhat在浏览器中分析堆转出快照，jhat dump.hprof -port 7000

- 从测试到上线：如何分析JVM运行状况及合理优化

  自行估算系统每秒大概多少请求，每个请求会创建多少对象，占用多少内存，机器应该选用什么样的配置，年轻代应该给多少内存，Young GC触发的频率，对象进入老年代的速率，老年代应该给多少内存，Full GC触发的频率，根据预估，先给自己的系统设置一些初始性的JVM参数，比如堆内存大小，年轻代大小，Eden和Survivor的比例，老年代的大小，大对象的阈值，大龄对象进入老年代的阈值，等等。优化思路其实简单来说就一句话：尽量让每次Young GC后的存活对象小于Survivor区域的50%，都留存在年轻代里。尽量别让对象进入老年代。尽量减少Full GC的频率，避免频繁Full GC对JVM性能的影响。

- 内存泄漏

  系统创建了大量的对象占用了内存，其实很多对象是不需要使用的，而且还无法回收掉

- Metaspace区域内存溢出

  第一种原因，很多工程师他不懂JVM的运行原理，在上线系统的时候对Metaspace区域直接用默认的参数，即根本不设置其大小这会导致默认的Metaspace区域可能才几十MB而已，此时对于一个稍微大型一点的系统，因为他自己有很多类，还依赖了很多外部的jar包有有很多的类，几十MB的Metaspace很容易就不够了。推荐的值在512MB

  第二种原因，就是很多人写系统的时候会用cglib之类的技术动态生成一些类，一旦代码中没有控制好，导致你生成的类过于多的时候，就很容易把Metaspace给塞满，进而引发内存溢出

- 堆内存溢出

  1. 系统承载高并发请求，因为请求量过大，导致大量对象都是存活的，所以要继续放入新的对象实在是不行了，此时就会引发OOM系统崩溃

  2. 系统有内存泄漏的问题，就是莫名其妙弄了很多的对象，结果对象都是存活的，没有及时取消对他们的引用，导致触发GC还是无法回收，此时只能引发内存溢出，因为内存实在放不下更多对象了

- JVM参数模板

  -Xms4096M -Xmx4096M -Xmn3072M -Xss1M -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFaction=92 -XX:+UseCMSCompactAtFullCollection -XX:CMSFullGCsBeforeCompaction=0 -XX:+CMSParallelInitialMarkEnabled -XX:+CMSScavengeBeforeRemark -XX:+DisableExplicitGC -XX:+PrintGCDetails -Xloggc:gc.log -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/usr/local/app/oom

- 对象创建

  （1）当Java虚拟机遇到一条字节码new指令时， 首先将去检查这个指令的参数是否能在常量池中定位到一个类的符号引用， 并且检查这个符号引用代表的类是否已被加载、 解析和初始化过。 如果没有， 那必须先执行相应的类加载过程。

  （2）在类加载检查通过后， 接下来虚拟机将为新生对象分配内存。 对象所需内存的大小在类加载完成后便可完全确定，为对象分配空间的任务实际上便等同于把一块确定大小的内存块从Java堆中划分出来。假设Java堆中内存是绝对规整的， 所有被使用过的内存都被放在一边， 空闲的内存被放在另一边， 中间放着一个指针作为分界点的指示器， 那所分配内存就仅仅是把那个指针向空闲空间方向挪动一段与对象大小相等的距离， 这种分配方式称为“指针碰撞”。但如果Java堆中的内存并不是规整的， 已被使用的内存和空闲的内存相互交错在一起， 那就没有办法简单地进行指针碰撞了， 虚拟机就必须维护一个列表， 记录上哪些内存块是可用的， 在分配的时候从列表中找到一块足够大的空间划分给对象实例， 并更新列表上的记录， 这种分配方式称为“空闲列表”（Free List） 。选择哪种分配方式由Java堆是否规整决定， 而Java堆是否规整又由所采用的垃圾收集器是否带有空间压缩整理（Compact） 的能力决定。 因此， 当使用Serial、 ParNew等带压缩整理过程的收集器时， 系统采用的分配算法是指针碰撞， 既简单又高效； 而当使用CMS这种基于清除（Sweep） 算法的收集器时， 理论上[1]就只能采用较为复杂的空闲列表来分配内存。

  （3）除如何划分可用空间之外， 还有另外一个需要考虑的问题： 对象创建在虚拟机中是非常频繁的行为， 即使仅仅修改一个指针所指向的位置， 在并发情况下也并不是线程安全的， 可能出现正在给对象A分配内存， 指针还没来得及修改， 对象B又同时使用了原来的指针来分配内存的情况。 解决这个问题有两种可选方案： 一种是对分配内存空间的动作进行同步处理——实际上虚拟机是采用CAS配上失败重试的方式保证更新操作的原子性；另外一种是把内存分配的动作按照线程划分在不同的空间之中进行， 即每个线程在Java堆中预先分配一小块内存， 称为本地线程分配缓冲（Thread Local AllocationBuffer， TLAB） ， 哪个线程要分配内存， 就在哪个线程的本地缓冲区中分配， 只有本地缓冲区用完了， 分配新的缓存区时才需要同步锁定。 虚拟机是否使用TLAB， 可以通过-XX： +/-UseTLAB参数来设定。

  （4）内存分配完成之后， 虚拟机必须将分配到的内存空间（但不包括对象头） 都初始化为零值， 如果使用了TLAB的话， 这一项工作也可以提前至TLAB分配时顺便进行。 这步操作保证了对象的实例字段在Java代码中可以不赋初始值就直接使用， 使程序能访问到这些字段的数据类型所对应的零值。

  （5）Java虚拟机还要对对象进行必要的设置， 例如这个对象是哪个类的实例、 如何才能找到类的元数据信息、 对象的哈希码（实际上对象的哈希码会延后到真正调用Object::hashCode()方法时才计算） 、 对象的GC分代年龄等信息。 这些信息存放在对象的对象头（Object Header） 之中。 根据虚拟机当前运行状态的不同， 如是否启用偏向锁等， 对象头会有不同的设置方式。

  （6）在上面工作都完成之后， 从虚拟机的视角来看， 一个新的对象已经产生了。 但是从Java程序的视角看来， 对象创建才刚刚开始——构造函数， 即Class文件中的<init>()方法还没有执行， 所有的字段都为默认的零值， 对象需要的其他资源和状态信息也还没有按照预定的意图构造好。 一般来说（由字节码流中new指令后面是否跟随invokespecial指令所决定， Java编译器会在遇到new关键字的地方同时生成这两条字节码指令， 但如果直接通过其他方式产生的则不一定如此） ， new指令之后会接着执行<init>()方法， 按照程序员的意愿对对象进行初始化， 这样一个真正可用的对象才算完全被构造出来。

- 逃逸分析

逃逸分析的基本原理是： 分析对象动态作用域， 当一个对象在方法里面被定义后， 它可能被外部方法所引用， 例如作为调用参数传递到其他方法中， 这种称为方法逃逸； 甚至还有可能被外部线程访问到， 譬如赋值给可以在其他线程中访问的实例变量， 这种称为线程逃逸； 从不逃逸、方法逃逸到线程逃逸， 称为对象由低到高的不同逃逸程度。

如果能证明一个对象不会逃逸到方法或线程之外（换句话说是别的方法或线程无法通过任何途径访问到这个对象） ， 或者逃逸程度比较低（只逃逸出方法而不会逃逸出线程） ， 则可能为这个对象实例采取不同程度的优化， 如：

栈上分配（Stack Allocations） ： 在Java虚拟机中， Java堆上分配创建对象的内存空间几乎是Java程序员都知道的常识， Java堆中的对象对于各个线程都是共享和可见的， 只要持有这个对象的引用， 就可以访问到堆中存储的对象数据。 虚拟机的垃圾收集子系统会回收堆中不再使用的对象， 但回收动作无论是标记筛选出可回收对象， 还是回收和整理内存， 都需要耗费大量资源。 如果确定一个对象不会逃逸出线程之外， 那让这个对象在栈上分配内存将会是一个很不错的主意， 对象所占用的内存空间就可以随栈帧出栈而销毁。 在一般应用中， 完全不会逃逸的局部对象和不会逃逸出线程的对象所占的比例是很大的， 如果能使用栈上分配， 那大量的对象就会随着方法的结束而自动销毁了， 垃圾收集子系统的压力将会下降很多。 栈上分配可以支持方法逃逸， 但不能支持线程逃逸。

标量替换（Scalar Replacement） ： 若一个数据已经无法再分解成更小的数据来表示了， Java虚拟机中的原始数据类型（int、 long等数值类型及reference类型等） 都不能再进一步分解了， 那么这些数据就可以被称为标量。 相对的， 如果一个数据可以继续分解， 那它就被称为聚合量（Aggregate） ， Java中的对象就是典型的聚合量。 如果把一个Java对象拆散， 根据程序访问的情况， 将其用到的成员变量恢复为原始类型来访问， 这个过程就称为标量替换。 假如逃逸分析能够证明一个对象不会被方法外部访问， 并且这个对象可以被拆散， 那么程序真正执行的时候将可能不去创建这个对象， 而改为直接创建它的若干个被这个方法使用的成员变量来代替。 将对象拆分后， 除了可以让对象的成员变量在栈上（栈上存储的数据， 很大机会被虚拟机分配至物理机器的高速寄存器中存储） 分配和读写之外， 还可以为后续进一步的优化手段创建条件。 标量替换可以视作栈上分配的一种特例， 实现更简单（不用考虑整个对象完整结构的分配） ， 但对逃逸程度的要求更高， 它不允许对象逃逸出方法范围内。

同步消除（Synchronization Elimination） ： 线程同步本身是一个相对耗时的过程， 如果逃逸分析能够确定一个变量不会逃逸出线程， 无法被其他线程访问， 那么这个变量的读写肯定就不会有竞争，对这个变量实施的同步措施也就可以安全地消除掉。